Towards a Formalization of the X86 Instruction Set Architecture

Sandip Ray

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712. USA.
sandip@cs.utexas.edu
http://www.cs.utexas.edu/users/sandip

Abstract

We present a preliminary approach to defining a formal specification of the semantics of the X86 Instruction Set Architecture. The goal of the formalization is to support the dual requirements of analyzing the correctness of binaries executing on the architecture and investigating different safety and security properties of the architecture itself. In particular, we focus on the security properties of protection rings available in the X86. A simplified version of the specification has been developed in the formal logic of the ACL2 theorem prover together with a generic approach to operationally define security policies. we discuss the use of our approach in developing trusted applications.

1 Introduction

Many modern system applications contain military and commercial secrets. It is important to ensure security properties of such applications, namely that no unauthorized interference or eavesdropping can occur. However, the complexity of computing systems makes it non-trivial to validate such properties either by code inspection or by simulation and testing. Formal, mechanized reasoning offers a promising alternative. In this approach, one models the system in some mathematical logic and derives its desired properties as theorems about the model, using a mechanical reasoning tool to assist in the verification process. When applicable it provides a mathematical guarantee for the desired properties of the system execution up to the accuracy of the model and the soundness of the reasoning tool. The goal of our work is to develop mechanical reasoning technology to facilitate analysis of security properties of systems based on the IA32 instruction set architecture.

The IA32 architecture, often colloquially referred to as the X86, is one of the most popular architectures used in microprocessor design, and is used ex-

tensively for developing critical software both at the system and the application levels. The protection mechanisms afforded by the underlying architecture therefore impacts the level of assurance that can be guaranteed by such software.

The protection mechanism provided by the X86 is the so-called *protection rings*, which attempts to distinguish between codes running with different privileges. The mechanism recognizes four privilege levels, numbered 0 to 3, where a greater number means less privilege. The code modules at a lower privilege level can access modules of higher privilege segments by means of a tightly controlled and protected interface called *gate*; attempts to access a higher privilege level without passing through the protection gate results in an exception. The ring mechanism enables development of a secure framework in which the code supporting the architectural capabilities of the security services execute at a higher privilege level than untrusted applications. The key properties we would like to ensure for such a secure framework include the following properties.

Tamper resistance: There is no way for code to modify data/code of a higher privilege level.

Non-bypassability: All accesses to system resources must be validated.

Our interest is in formal analysis of such properties for the secure ring architecture. To this end, we use the ACL2 theorem prover and its mechanical reasoning engine [5] to formalize and reason about the X86 architecture. ACL2 is an industrial-strength theorem prover for a first-order logic of recursive functions, and has been used to reason about some of the largest computing systems designs ever to undergo formal analysis [3, 11, 8]. Our goal is to develop techniques to reason about the ring architecture in ACL2 with reasonable automation. In addition, practical limitations require that the methodology be amenable to the following:

Incremental reasoning: The X86 architecture is extremely complex. The specification released by Intel Corporation [1] consists of 5 volumes each about 500 pages. Given the size and complexity of the architecture, it is impractical to define a relatively complete formal definition of its specification in a single iteration. The formal specification and reasoning process therefore must enable an incremental reasoning approach, whereby on can start with a relatively simple machine model which can be iteratively refined while preserving reuse of formal results achieved on the abstraction.

Decoupling: Formal verification of security of the ring architecture entails a formal definition of the security policy enforced by the architecture. Once such a policy has been formalized and verified, software running on top of the architecture can enforce their own security mechanisms, assuming the security of the protection mechanisms of the underlying architecture. To facilitate this and enable reasoning about such software, it is desirable to formalize the security policy as an intermediate abstraction insulating the definition of the architecture implementation from the software-level view of the protection mechanisms. This also provides the benefit of decoupling

the security properties of the software from the the underlying processor implementation.

In this paper, we discuss our progress in the development of such a verification technology. To facilitate incremental reasoning, we develop a two-pronged approach to formally modeling the X86 ISA. We model the architecture at two different levels of abstraction. At the lower level model, the formal definition of the ISA accurately reflects the specifications described in the X86 reference manual [1]. We have defined the semantics of 14 X86 instructions at this level; the model is so detailed that it can execute X86 binaries on the supported instructions generated by compiling them from high-level C programs by a standard `gcc` compiler. On the other hand, the details and complexity of this model make reasoning about the protection mechanisms itself complex. We therefore develop a much simpler model that defines the formalization at just enough detail to facilitate such reasoning, while still maintaining a clear correspondence with the lower-level model. The formalization of the intermediate abstraction thereby factors our verification complexity, enabling us to focus on the different aspects of the design at different levels.

To achieve decoupling, we formalize the security policy of the ring architecture as an abstract state machine constrained to respect the privilege levels of the underlying architecture. We achieve this using the encapsulation mechanism of the ACL2 theorem prover [2] which enables us to specify a function by merely axiomatizing certain properties. The security policy can be used by a software to implement its own security mechanism without concern for the processor implementation. Furthermore, we develop mechanical proofs with ACL2 showing that under certain conditions, the security policy ensures non-bypassability, and also that the simplified processor model respects the policy.

More concretely, the following are the key contributions from this project:

1. A formal accurate model of a small subset of the X86 architecture.
2. A simplified processor model for studying the ring architecture and its protection mechanisms.
3. A formal definition of the security policy enforced by the ring architecture.
4. A mechanical proof of security requirements for our simplified processor model.

In the remainder of the paper, we discuss these contributions in some detail. We discuss some aspects of our processor models in Section 2 and the security policy in Section 3. In Section 4 we lay out the basic approach to verify the security properties of the processor design. We conclude in Section 5 with a discussion of future directions.

2 Processor Models

In this section, we discuss the two levels of ISA models for the X86, that we have developed in ACL2. Both models are defined operationally; that is, we define a representation of the processor state as a formal object (tuple) in the ACL2 logic and define how each instruction manipulates this formal object [10]. Thus the formal models also act as an executable simulator for the processor architecture. The key difference between the two processor models is in the level of detail at which the different state components are modeled and the updates specified by the formal definition.

2.1 A Low-level X86 Specification

At the lowest level we model the processor to accurately reflect the specifications in the Intel X86 manual [1]. Only a small subset of 14 instructions has been modeled currently at this level; the effect of each instruction is defined by formalizing its effect on the machine state. The state is formalized as a record [7] which represents the evaluation of the program registers (`EAX`, `ECX`, `EDX`, `EBX`, `ESI`, `EDI`, `ESP`, and `EBP`, the instruction pointer `EIP`, the `EFLAGS` register, the segment registers, and the memory. The instruction format is also faithful to the manual, although we do not model the 64-bit modes. For each instruction *inst*, we define a function `execute-`*inst* that captures its effect. To illustrate the detail at which the instructions are modeled, we show a tiny fragment of the definition of the `execute-ADD` instruction:

```
(defun execute-ADD (inst s)
  (let* ((opcode (getByte 0 inst))
         (ModR/M (getByte 8 inst)))
    (case (hex opcode)
      ;; 04 ADD AL,imm8    - Add imm8 to AL
      (04
       (let* ((imm       (imm8 (- (len inst) 8)
                               inst))
              (AL        (g :AL s))
              (nat-AL    (bv-to-nat AL))
              (nat-imm   (bv-to-nat imm))
              (int-AL    (bv-to-int AL))
              (int-imm   (bv-to-int imm))
              (nat-sum   (+ nat-AL nat-imm))
              (nat-result (nat-to-bv 8 nat-sum))
              (int-sum   (+ int-AL int-imm))
              (result    (int-to-bv 8 int-sum))
              (ZF        (if (equal int-sum 0) T
                           NIL))
              (OF        (getOF result int-sum))
              (SF        (car result))
```

```
            (num-one   (t-count result))
            (PF        (if (evenp num-one) T
                         NIL))
            (CF        (getCF nat-result
                              nat-sum))
            (AF        (getAF AL imm))
            (eip       (g :eip s))
            (nat-eip   (bv-to-nat eip))
            (len-inst  (/ (len inst) 8))
            (new-eip   (+ nat-eip len-inst))
            (next-eip  (nat-to-bv 32
                                  new-eip)))
        (>s :AL result
            :ZF zf
            :PF pf
            :OF of
            :SF sf
            :CF cf
            :AF af
            :EIP next-eip)))
     ;; 05 ADD EAX,imm32    - Add imm32 to EAX
      (05 ...)
    ...)))
```

We have skipped most of the formal definitions and showed only a fragment of the definition, but this should provide an idea of the details of the model. Using these operational definitions of the individual instructions, we model the state transition of the ISA as a function that fetches and executes one instruction (pointed to by the `EIP`) at each step. The semantics of the 14 instructions together constitute about 5000 lines of formal definitions in ACL2, in addition to generic libraries of bit vector operations used to support the definitions. The model can act as a simulator for X86 binaries, and has been used to simulate binaries directly generated from high-level C-code by the `gcc` compiler for simple programs such as the factorial and Fibonacci computations.

2.2 A More Abstract Model

The low-level model above is a step towards defining a faithful formalization of the X86 ISA. However, the complexity of the model makes the direct verification of security properties difficult and cumbersome. To factor out the complexity, we define a much simpler abstract ISA, while still preserving correlation with this complex architecture. This enables us to make progress on the security verification by focusing on the protection mechanisms of the ISA without getting distracted by extraneous details.

Our abstract model is inspired by the Y86 processor developed at Carnegie-Mellon University by Bryant and O'Hallaron [4]. Although inspired by the X86,

the processor has fewer data types and instructions. In addition, some of the instructions are simplified; for instance the `MOVL` instruction is split into four instructions `RRMOVL`, `IRMOVL`, `RMMOVL`, and `MRMOVL`, explicitly representing the source and the destination of the operation. The simplicity of the model makes it an ideal medium for extension to study and analyze the memory protection mechanisms of the architecture.

We have extended the basic Y86 processor to formalize protection mechanisms. Our model (referred to as Y86$^+$) defines two modes of operation (*real-address* and *protected*), supports segmentation, Local and Global Descriptor Tables, call-gate, and near and far procedure calls.[1] Some exceptions (for instance general protection exception) are formalized, and we formalize segment access rights, four privilege levels of the ring architecture, and segment conformance. The processor faithfully models the Intel specification with respect to memory references. Nevertheless, the model is severely simplified. SMM, PM, and virtualization mechanisms are ignored. The processor description, developed operationally as a formal simulator, involves about 1500 lines of formal definition.

3 Security Policy

Recall that the key focus of our work is in the verification of the security properties of the ring architecture. Reasoning about security requires a precise, unambiguous characterization of the security policy enforced by the architecture. We now describe how we develop an abstract specification of the security policy in the logic of ACL2.

The key logical feature we use is *encapsulation*. Encapsulation is an extension principle which allows definition of partial functions by specifying a collection of constraints. For instance, we can constrain a function axiomatized merely to return a natural number (without specifying the return value). Kaufmann and Moore [6] show that the encapsulation principle introduces axioms that are conservative. Furthermore, ACL2 provides a derived inference rule called functional instantiation [2] which enables one to lift a theorem about constrained functions to concrete functions which satisfy the constraints.

The formal security policy is modeled by a collection of constrained functions. Here we describe some of the functions and their intuitive semantic descriptions. Here `st` represents the current processor state.

- `(segs st)` returns the set of memory segments.
- `(current-segment st)` returns the currently executing segment.
- `(eip st)` returns the instruction pointer.
- `(privilege st)` returns the current access privileges and is constrained to return a number between 0 and 3.

[1] In our current model we only support segmentation, but not paging. We are extending the model to incorporate paging.

- `(next st)` returns the next processor state.
- `(accessible-p inst st privilege)` holds if the `privilege` permits execution of the memory segment called by `inst` (if `inst` is a (far) call instruction).
- `(inv st)` is an invariant of the current state.
- ...

The set of constraints precisely characterize the conditions under which the processor enables execution of the current instruction. For instance one of the constraints is specified below:

```
(let ((inst (current-inst (fetch (eip st) (current-segment st)))))
  (implies (and (call-instp inst)
                (inv st)
                (not (accessible-p inst st (privilege st)))
           (exception-p (next st))))
```

The constraint says that if the current instruction `inst` (fetched from the current segment from the address pointed to by the `eip` is a (far) call instruction and the destination of `inst` is not accessible according to the current privileges then the transition causes an exception.

The security policy involves 15 constrained functions and about 36 constraints similar to above. The policy is inspired by and similar in spirit to the separation kernel security policy developed for the Rockwell-Collins AAMP7^{TM} separation kernel security policy, but is much more elaborate to account for the eccentricities of the X86 architecture. Nevertheless the policy completely insulates the state representation of the underlying models.

4 Mechanical Reasoning

Mechanical reasoning on the processor models has focused on four main thrust areas:

1. Determine conditions under which the security policy outlined above insures tamper-resistance and non-bypassability.
2. Determine the key invariants under which the Y86$^+$ model implements the security policy.
3. Develop techniques for reasoning about software running on the low-level X86 model.
4. Mechanically relate the low-level X86 model with Y86$^+$.

We now outline our progress in the above thrust areas.

- We have shown that the security policy above guarantees non-bypassability under a certain invariant. We have also defined a predicate which guarantees that $Y86^+$ implements the security policy. We are currently analyzing conditions under which the predicate defined is an invariant, which will complete the proof of non-bypassability of $Y86^+$. We are also analyzing the invariants under which the security policy guarantees tamper-resistance. We anticipate to complete these proofs in the near future. Tamper-resistance has already been proven for a slightly stronger security policy which involves two (rather than four) privilege levels as necessary for the ring architecture; we plan to lift this proof to the full ring.

- We have recently completed a methodology based on cutpoints to automate proofs of JVM bytecodes by symbolic simulation [9]. We are adapting the approach to the X86 model.

- We are developing a formal characterization of trace equivalence to relate the high and low-level X86 model. This is currently in very early stages of understanding. However, since trace equivalence preserves security properties of execution traces we believe a proper formalization will enable us to lift $Y86^+$ proofs to the low-level X86 model.

5 Conclusion and Future Work

We have outlined initial progress in formally reasoning about security properties of the X86 ring architecture. To this end, we have formalized an abstract model $Y86^+$ and an encapsulated security policy. We have proven that the security policy guarantees non-bypassability and identified invariants in $Y86^+$ which must be preserved to demonstrate that the processor implements the security policy. We have also been working on developing a low-level accurate model of the X86 ISA and are investigating approaches to show formal correspondence between this low-level model and $Y86^+$.

The work reported is a result of about three months of research by the author working about 60% of time on the project. The initial results indicate that it is possible to analyze the security properties of the X86 ring architecture with reasonable effort. Nevertheless it should be clarified that we have only scratched the surface so far. The X86 model is one of the most elaborate processor models and substantial work remains both to formally specify the architecture and lift our results to this model. We are currently working on proving tamper-resistance on $Y86^+$. Eventually we anticipate refining and elaborating the low-level X86 model and lifting the security proofs to this model by showing a correspondence with $Y86^+$.

Acknowledgements

The research has been supported in part by DARPA and National Science Foundation under Grant No. CNS-0429591. Matt Kaufmann provided numerous insights including suggestion of a formal approach to formalize properties like non-bypassability in ACL2. Ravi Kolli did initial work on developing the detailed model described in Section 2.1.

References

[1] IA-32 Intel Architecture Software Developers' Manual. See URL `http://-www.intel.com/design/pentium4/manuals/index_new.htm`.

[2] R. S. Boyer, D. Goldshlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[3] B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of the* 1*st International Conference on Formal Methods in Computer-Aided Design (FMCAD 1996)*, volume 1166 of *LNCS*, pages 275–293. Springer-Verlag, 1996.

[4] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2003.

[5] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[6] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[7] M. Kaufmann and R. Sumners. Efficient Rewriting of Data Structures in ACL2. In D. Borrione, M. Kaufmann, and J S. Moore, editors, *Proceedings of* 3*rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*, pages 141–150, Grenoble, France, April 2002.

[8] H. Liu and J S. Moore. Executable JVM Model for Analytical Reasoning: A Study. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines, and Emulators*, San Diego, CA, June 2003.

[9] J. Matthews, J S. Moore, S. Ray, and D. Vroon. Verification Condition Generation via Theorem Proving. In M. Hermann and A. Voronkov, editors, *Proceedings of* 13*rd International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, November 2006.

[10] J. McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Congress*, volume 62, pages 21–28. North-Holland, August 1962.

[11] J. Sawada and W. A. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.